

High Message Rate, NIC-Based Atomics: Design and Performance Considerations

Keith D. Underwood ^{#1}, Michael Levenhagen ^{*2}, K. Scott Hemmert ^{*3}, Ron Brightwell ^{*4}

[#] *DEG Architecture and Planning, Intel Corporation*
MS-1319, P.O. Box 5800, Albuquerque, NM, 87185-1319 USA

¹ *Keith.D.Underwood@intel.com*

^{*} *Sandia National Laboratories*
MS-1319, P.O. Box 5800, Albuquerque, NM, 87185-1319 USA

² *mjleven@sandia.gov*

³ *kshemme@sandia.gov*

⁴ *rbbrigh@sandia.gov*

Abstract—Remote atomic memory operations are critical for achieving high-performance synchronization in tightly-coupled systems. Previous approaches to implementing atomic memory operations on high-performance networks have explored providing the primitives necessary to achieve low latency and low host processor overhead. In this paper, we explore the implementation of atomic memory operations with a focus on achieving high message rate while maintaining these other desirable characteristics. We believe that high message rate is a key performance characteristic that will determine the viability of a high-performance network to support future multi-petascale systems, especially those that expect to employ a partitioned global address space (PGAS) programming model. As an example, many have proposed using network interface level atomic operations to enhance the performance of the HPCC RandomAccess benchmark. This paper explores several issues relevant to the design of an atomic unit on the network interface. We explore the implications of the size of the cache as well as the associativity. Given the growing ratio of bandwidth to latency of modern host interfaces, we explore some of the interactions that impact the concurrency needed to saturate the interface.

I. INTRODUCTION

The constant advancement of silicon technology in accordance with Moore’s Law has provided dramatic advances in microprocessor performance. At the same time, it has opened the door to extending the functionality of the network interface in high performance computing systems. For example, the Cray SeaStar[1] [2] and Quadrics Elan[3] [4] interconnects include processors that are capable of offloading message processing. Recent proposals have included other hardware constructs to enhance the performance of message passing offload[5] [6].

Another common construct on high-end network interfaces is a hardware unit to provide low latency, remote atomic memory operations. Atomic memory operations form the basis of numerous approaches to synchronization[7] and, as such, are important to the successful use of the emerging partitioned

global address space (PGAS) programming model. Languages like UPC[8] and Co-array Fortran[9] enable fine-grained communications, which ultimately increase the need for fine-grained synchronization and the performance of atomic operations upon which they are built. As the number of cores in high end systems grows exponentially in the multi-core era — perhaps reaching one million cores in near term petascale systems — the rate of atomic memory operations will become as critical as the latency. With the bandwidth of modern systems growing far more rapidly than latency is decreasing, several challenges arise for the design of an atomic memory unit on the NIC. Challenges include how to design a pipelined atomic unit that exposes concurrency without exposing data hazards as well as how to implement an efficient NIC side cache to cover the latency to the host memory. This paper explores how to address these challenges with the architecture in Section III and the appropriate size and organization of an atomic unit cache.

A second motivation for an atomic unit on the network interface is the type of operations represented by the HPC Challenge (HPCC) RandomAccess benchmark. While the Linpack benchmark[10] has long been the basis of the Top 500 listing[11], it has been identified as overly biased toward one feature of system performance (dense matrix floating-point arithmetic). In response, the broader HPC Challenge (HPCC) Benchmark Suite[12] has been introduced to broaden the assessment of system parameters to range from memory bandwidth (STREAM) to network performance. One particularly interesting member of the HPCC Benchmark Suite is the RandomAccess benchmark, which embodies the longer-standing “Giga-Updates Per Second (GUPS)” benchmark.

The RandomAccess benchmark generates a series of random integers that specify system-wide memory locations to update; thus, each update is an 8-byte XOR to a random memory location on a random node. To allow system implementers some flexibility while attempting to maintain the spirit of the benchmark, the RandomAccess benchmark allows up to 1024 outstanding updates per thread. The result is small, random updates that test a variety of network and system char-

[#]This work was performed while Keith Underwood was a staff member at Sandia National Labs.

^{*}Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

acteristics, including the rate at which the network interface can handle small messages, the overhead of the network, and the overhead imposed by caching systems on the node. The relatively high overhead and low message rate of traditional distributed memory MPPs has led to novel implementations of the benchmark on high-end systems [13], [14] that were designed to better aggregate messages. If, however, sufficient message rate is provided by the platform, the benchmark can perform quite well using a baseline-like implementation[15].

An alternative mechanism to accelerate the RandomAccess benchmark is to use a unit on the network interface (NIC) that performs atomic operations. Such a unit would avoid the typical need to write incoming messages into memory — a process that consumes a substantial amount of precious memory bandwidth. However, the RandomAccess benchmark is extremely sensitive to *message rate* and an atomic unit on the NIC will require that items to be updated be read by the network interface and consume precious bus bandwidth. As the correct solution is not obvious, this paper examines the trade-offs associated with moving the operations associated with the RandomAccess benchmark to the network interface, along with some of the relevant atomic unit design parameters.

In the next section, we present previous work in the area. Following that, Section III presents the atomic unit that was modeled and Section IV presents the simulation methodology. The results are presented in Section V, and then compared to theoretical bounds in Section VI. Finally, conclusions are presented in Section VII and the paper closes with future work in Section VIII.

II. BACKGROUND

Atomic memory operations (AMOs), such as fetch-and-increment and compare-and-swap, are a fundamental capability for shared memory processors that provide scalable synchronization primitives for cooperating processes [7]. Shared memory-based parallel processing machines have provided this capability as a fundamental component of the memory and caching subsystem.

In the early 1990's, as the popularity of distributed shared memory (DSM) machines [16] began to increase, remote AMOs were implemented on several machines, such as the Cray T3 [17] series and the Meiko CS-2 [18], to enable efficient synchronization and coordination of processes in a networked environment. The Cray SHMEM [19] library was one of the first popular network programming interfaces to support a variety of remote memory AMOs and expose this capability to the user. See [20] for a complete discussion of the implementation and evaluation of AMOs on several DSM systems. Today, large-scale global shared memory machines, such as the Cray X1 [21] and Black Widow [22], carry on this tradition of supporting AMOs in hardware.

Commodity high-performance networks that followed these proprietary systems also began to include support for remote AMOs. Networks such as the Quadrics Elan [3], Scalable Coherent Interface [23], and, more recently, InfiniBand [24] have included hardware support at the network interface level

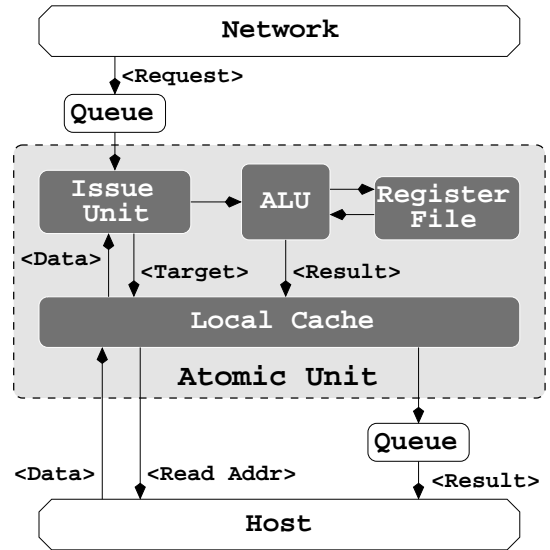


Fig. 1. Block diagram of the atomic unit

for implementing remote AMOs. In addition, the programmability of the Myrinet [25] network allowed for an efficient implementation of remote AMOs by enhancing the firmware running on the network adapter [26].

This paper is unique in that it is the first effort to explore *high message rate* AMOs in a modern system context, where the bandwidth to the host processor continues to grow and the latency is rapidly approaching a lower bound. As such, we focus on issues of exposing concurrency and implementing caching in such a way as to allow a stream of AMOs to approach the bandwidth of the host bus. This requires care in the design of both the AMO functional unit as well as the cache on the network interface.

III. ATOMIC UNIT ARCHITECTURE

This section describes the high-level architecture of the atomic unit as shown in Figure 1. Atomic operations received by the NIC are directed to the atomic unit (routing path not shown in the figure), which makes a request of the local cache for the data value to update. If the request misses the cache, the cache requests the data from the host. The ALU then updates the value and writes the result back to the local cache. If requested, the updated value is also returned over the network to the requester.

The local cache is configured in write-through mode: every value written to the cache from the functional unit is immediately propagated to host memory. While this sounds like an unusual configuration (most processors use a write-back cache to obtain higher performance), it is actually intended to simplify the NIC-to-host interaction while maintaining performance. For example, a local read of a local variable that is updated by the atomic unit should only go to local memory. A write-through cache provides immediate updates to the host and eliminates the need to have something like a timer per cache line to ensure that the line is eventually visible to the

host. However, it is still desirable to reduce the bandwidth requirements on the host link, when possible. To accomplish this, a rate-limiting atomic unit is used.

A. Rate Limiting Atomic Unit

Frequently, there is a small number of target addresses (e.g. locks) used for atomic operations. A configurable timeout on the cache could be used to leverage this fact to condense multiple atomic operations into a single host transaction. However, if the interval between two operations is never greater than the timeout, the result would never be written to the host. Switching to a write-through cache eliminates this challenge, but still allows the atomic unit to leverage the network access pattern.

The atomic unit operates by separating cache requests from the actual operations (see Figure 1). The issue unit translates operations into a stream of cache requests that cause the cache to generate a queue of values back to the issue unit. Once the data is available, the request can be issued to the ALU. This arrangement opens up many possibilities for limiting the rate at which data is written back to the cache, and therefore to the host. Specifically, host writes are limited by executing multiple operations that target the same address and arrive in close temporal proximity before writing the result to the cache. Three resources are needed to accomplish this. First, the issue unit is provisioned with logic to look at a window of requests (referred to as the look-ahead window) that have backed up at the input to the atomic unit. Second, a small register file is needed to hold intermediate values. Third, a method for tracking busy cache entries is needed. A cache entry is busy if it has been read and not yet written back (either to the cache or the register file).

To condense the host traffic, the issue unit is allowed to rewrite the source or destination of an atomic request. A write to the register file (rather than the cache) is made if an operation in the look-ahead window targeted the same address as the current operation. A later instruction targeting the same address would then use the temporary register as a source and write the result back to the cache, thus causing a write to the host. Once an instruction is issued, it cannot be changed; therefore, the tracking of busy cache entries becomes necessary. This is necessary to ensure that the required data is ready to be read and is not still in-flight. If the data is unavailable, the issue unit blocks. However, it would be possible to include a set-aside buffer for such operations to allow ready operations to proceed.

It is generally desirable to use an “aging” policy to determine how long a value can be reused from the register file before being written to the cache. One aging policy is simply to limit the number of times a single target can be written to the register file before being sent to cache. A better policy is to allow the issue unit to monitor the status of the queue to the host. If the queue to the host is *empty* (or nearly so), then the issue unit will instruct the ALU to write the result back to the cache to generate a write to the host. If the queue to the host is backed up, then the host link is being over utilized, and

it is better to suppress the write by targeting the register file. It is also possible to provide a policy that is a combination of these. In all cases, a result is written to the cache if no other instruction in the lookahead window targets the same address.

This approach fundamentally differs from a timeout in that it simply provides a view into queued operations, whereas a timeout is designed to look much further into the future. The reliance on only queued operations recognizes that there is only a measurable bandwidth constriction when there is a queue of operations at the input to the function unit.

B. Resolving Read-after-Write (RAW) Hazards

In many systems, a cache for the atomic unit on the network interface may create a read-after-write (RAW) hazard. Specifically, the sequence: 1) perform an atomic operation on address A, 2) write the result to host memory at address A, 3) evict address A from the cache, and 4) perform a second atomic operation on address A. Step 4 in this sequence will cause a host memory read after the host memory write from step 3. If those operations are temporally close, it is possible (even likely) for the read access to pass the previous read. This situation can easily occur with pipelined I/O interfaces that do not guarantee ordering of requests.

The solution to this problem involves a unit to buffer host writes until they have completed to host memory. This buffer works as a secondary cache structure. The purpose of this buffer is to ensure that writes to host memory complete to a level of the memory hierarchy where ordering is guaranteed before a read or write to/from the same address is issued. This secondary structure is then checked after a cache miss, before a read or write is issued. There are two main approaches to managing this buffer. The first approach is to require that the host bus interface return an acknowledgment when the write request has completed to a level of hierarchy that preserves request ordering. When the acknowledgment is received, the item can be deleted from the buffer. The second approach is to request a *flush* from the host bus interface (this is available on most interfaces). This flush would only return when all outstanding requests completed. In this scenario, it is not possible to track when individual writes complete; thus, only evicted items are placed in this buffer. When the buffer is full, a *flush* request is issued and the buffer is emptied. The first approach is generally preferred, as it requires a smaller buffer to provide good performance.

C. Pipelining

To sustain high performance under certain workloads (those that primarily miss in the cache, such as the HPCC RandomAccess benchmark), it is critical that the function unit and cache be pipelined to maintain a sufficient number of outstanding accesses to host memory to cover the round-trip latency. Pipelining of accesses to the host begins with the issue unit providing a stream of address requests to the cache. When a cache miss is encountered, the cache must forward the request to the host interface and attempt to service the next request. The issue unit associates a tag with each cache

request, allowing the results to be returned out-of-order, and making it possible for processing to continue on operands that hit in the cache. This is key to allowing enough host requests to be in-flight in the case where cache hits and misses are interspersed. Although the cache results can be returned out-of-order, instructions are issued to the function unit in-order. Depending on the ordering constraints imposed on the atomic operations, this could be relaxed and instructions could be issued out-of-order, but this would also increase the complexity of the atomic unit.

IV. METHODOLOGY

This work is based on the Structural Simulation Toolkit (SST) developed by Sandia National Laboratories and the University of Notre Dame[27], [15]. SST integrates the SimpleScalar[28] processor simulation into a hybrid discrete event and cycle-driven simulation infrastructure. This allows us to integrate relatively coarse-grained network interface models with relatively fine-grained processor simulations. To better replicate system behavior, SST adds a robust memory model, including the ability to memory map I/O devices, to the traditional SimpleScalar environment. SST has also been extended to model a HyperTransport I/O interface that allowed us to model both a Cray XT3 supercomputer[27] and a similar system exploring hardware support for PGAS[15].

TABLE I
PROCESSOR PARAMETERS

Parameter	CPU
Clock Frequency	2 GHz
Cores per Node	2
Fetch Queue	4
Issue Width	8
Commit Width	4
RUU Size	64
Integer Units	4
Memory Ports	3
L1 (Size/Assoc.)	64KB/2
L2	1MB/16
ISA	PowerPC
Main Memory Bandwidth	6.4 GB/s peak
Main Memory latency	140-160 cyc.
System I/O	HyperTransport
I/O Bandwidth	2.3 GB/s/dir sustainable
I/O Latency	250 ns

TABLE II
NETWORK PARAMETERS

Topology	3D Torus
Clock Frequency	500 MHz
Link Bandwidth (Peak)	4 GB/s/dir
Router Latency	50 ns
Link Overhead	15%
Router Arbitration	Round-Robin
Router VCs	4

To study the performance of collective operations, we used the model of a SHMEM network interface from [15]. Tables I and II are replicated here to present the salient properties of

the processor and network models. The HyperTransport (HT) interface is modeled as a 2.3 GB/s per direction link, before accounting for the “header flit” of each HyperTransport packet, based on tuning done for [27]. Similarly, the baseline latency of the HyperTransport interface is modeled as 250 ns, based on measured hardware performance. The host processor runs at 2 GHz and the network interface at 500 MHz. Both match the points validated in previous work [27].

Like the previous work, the router models synchronous links (8 bytes wide, 500 MHz) and accounts for the difference in peak bandwidth by adjusting the overhead (15% modeled versus the 24% protocol overhead seen in the real system). Finally, the virtual channel architecture (two virtual channel classes each having two virtual channels) is modeled along with round-robin arbitration.

This work explored two basic benchmarks. The first was a simple atomic benchmark. This benchmark used applications of N PEs, where $N-1$ PEs target PE0 with atomic operations. Each PE issued 1000 atomic operations, with either all atomics for all PEs targeting a single memory location or each PE targeting 16 unique memory locations, and then synchronized before completing the timing test. In the first case, this meant that all atomic operating at the target were to a single memory location. In the second case, this led to $(N-1) \times 16$ different memory location targets. The first variant of the benchmark is meant to mimic the type of behavior that might be seen when a single variable is highly contested for a scenario such as a lock, while the second variant is designed to show what may occur when there is less contention over the single memory location.

For RandomAccess benchmark, the baseline that was used was the work performed in [15]. This implementation leveraged the high performance SHMEM implementation to send updates to their respective processors and then used the processor to perform the local update. A second implementation was created using an 8 byte atomic XOR command to explore the improvements available with a hardware atomic unit.

V. RESULTS

A variety of configurations were explored to assess the performance potential of an atomic unit on the NIC. For each benchmark (the focused atomic benchmark and the RandomAccess benchmark), various design points were explored to determine whether the atomic unit should have a cache, how large that cache should be, and what its configuration should be. In addition, the impact of HyperTransport (HT) latency was explored as well as the impact of HT “tags”. In this case, “tags” refer to the identifiers used to track the outstanding transactions (and, therefore, the number of outstanding transactions) that the NIC can have on the HT link. For each scenario, the performance is presented in “MegaBytes per Second”. Each update is 8 bytes, so the update rate can be inferred from the data rate.

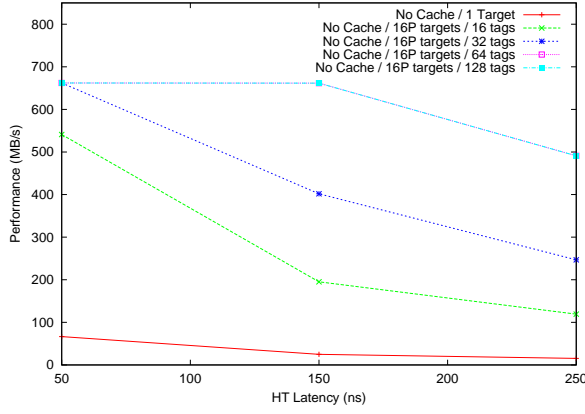


Fig. 2. The performance of an atomic unit in the absence of a cache

A. Atomic Unit Performance

The performance of the atomic unit was assessed based on two variants of a benchmark where all nodes but one target a single node. The first variant has all accesses target a single address, which is meant to mimic a highly contended lock variable. The second variant exposes more concurrency to the atomic unit by having each source node target 16 *unique* locations, so that there is no contention for a single address among the nodes and even the contention from a single node is limited. For the sake of comparison, all of the graphs in this section use the same scale.

In Figure 2, we see the performance the atomic unit for a 64 PE¹ system (32 dual-processor nodes) when *absolutely no cache* is provided. Thus, even if two sequential accesses are in the queue on the NIC, and both target the same memory address, the first update will be performed and written back to memory and then the second update will read the data from memory to perform the update. When all updates target a single location, this results in very poor performance indeed (the bottom line in the graph), as each update incurs the round-trip latency across the HT bus. In contrast, when more concurrency is available, because each source PE targets a variety of different memory addresses (the other three lines), the atomic unit can have substantially more requests pipelined to hide much of the bus latency.

Two important parameters are varied in Figure 2. First, along the X-axis, the latency of the HT bus is varied from 50 ns — an aggressive design point — to 250 ns — the value determined in [27]. The value of this latency has a substantial impact on the concurrency needed to hide the latency. The second important parameter varied in Figure 2 is the number of *tags* — or outstanding requests — that HT is allowed to have. The “typical” value for tags in a real HT implementation is 32, which is acceptable for the lowest latency scenario. At higher latency, 64 tags is clearly sufficient (the 64 and 128 lines completely overlap); however, at a 250 ns latency, a secondary bottleneck clearly constrains performance. We are

currently exploring where this bottleneck lies.

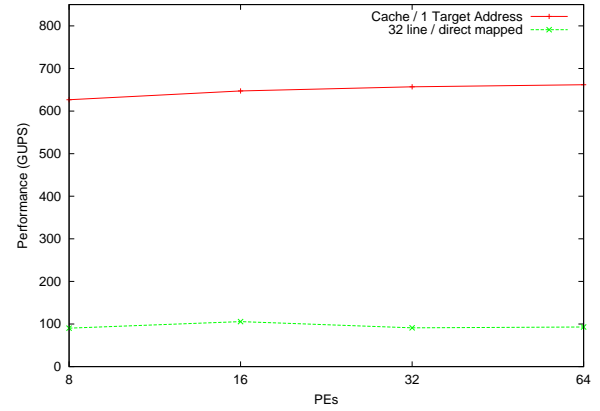


Fig. 3. The performance of an atomic unit with a cache

In Figure 3, a small cache (32-item, direct- mapped) is introduced in the atomic unit. This results in a substantial boost in the performance of the scenario where all incoming messages target a single memory location. In fact, the performance reaches the limit of the network bandwidth (a 48-byte message carrying 8 bytes of data results in a maximum atomic payload rate of $\frac{1}{6}$ of the maximum data rate). In stark contrast, the small cache offers only minimal benefit to the case where each node targets an array of unique addresses. This phenomena occurs because the substantial number of unique addresses being targeted causes most operations to miss the cache. For example, even at 8 PEs, 112 unique locations are targeted (7 PEs targeting 16 unique addresses each). Furthermore, the direct mapped nature of the cache makes it likely that two items will conflict for a cache line. Much like in a processor, when this scenario occurs, further accesses are blocked while the first access is serviced and then consumed. The second access must wait for the cache line to become available to provide a location for a returning request to be written.

To explore the impact of the size and configuration of the cache on the performance of the atomic unit, Figure 4 varies both the size (in lines) and the associativity of the cache². The cache size is along the X-axis to highlight the impact of cache size, while the various lines represent varying levels of contention based on the number of PEs used and the associativity of the cache. Overall, Figure 4 suggests that increasing the associativity of the cache is more important than increasing the size of the cache; however, there is a disturbing trend in the lines that suggest that the cache size needs to continue to grow with system size. Fortunately, this is not actually the case for two reasons.

Despite the appearances of Figure 4, the size of the cache does not inherently need to be linked to system size. Instead, the results can be explained by two phenomena. First, the number of addresses targeted by these tests is artificially large

¹All system sizes tested yielded the same performance.

²Note that the 32 line, 64-way caches are shown as “zero” performance, because that case is not possible

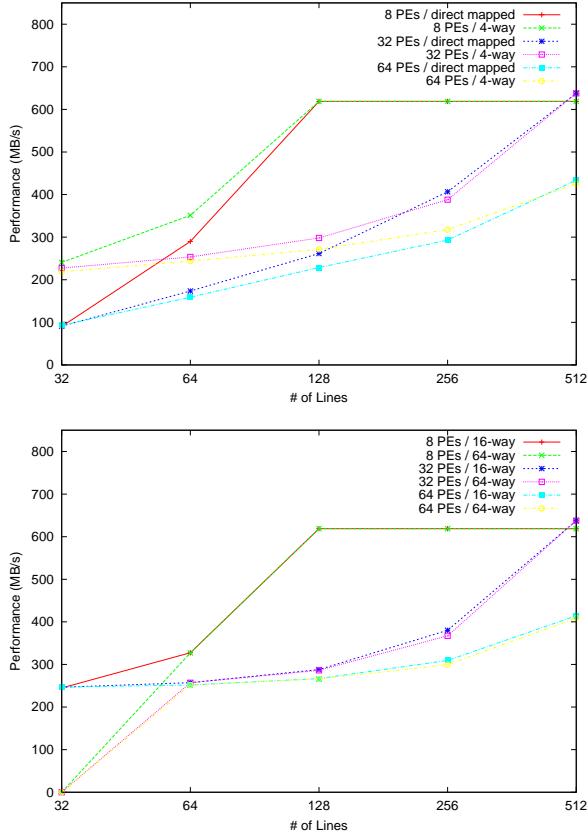


Fig. 4. The performance of an atomic unit using a cache of variable sizes and associativities

to facilitate exploring issues related to scaling. Applications are more likely to either target a large number of addresses (and, thus, look more like the RandomAccess results in Section V-B) or a much smaller number of addresses related to synchronization. In the latter case, which is the point of Figure 4, the number of synchronization variables on a *given target PE* is likely to be *related* to system size, but not *linear* with system size as illustrated here. Unfortunately, it is not practical for us to simulate systems of very large scale, so the linear growth in target addresses was used to explore issues of contention in an attempt to mimic systems of much larger size.

The primary reason for the performance issues seen in Figure 4, however, are explained by Figure 5³. On the simulated system, the performance of the network interface is well matched to the performance of the HyperTransport interface; thus, as long as there is sufficient *concurrency* to tolerate the latency of the HT link, it should be possible to sustain the full bandwidth of the network. In Figure 4, the full bandwidth of the network is only sustained when many of the atomic updates hit in the cache. Cache misses may occur due to evictions forced by conflict due to limited associativity or due to capacity based evictions; however, even with frequent

³Again, note that the “zero” performance of the 32 line, 64-way cache is due to the fact that this configuration is impossible.

cache misses, it should be possible to sustain most of the network bandwidth as long as *concurrency* is not constrained. In Figure 3, we saw concurrency constraints from the direct mapped nature of the cache; however, the contrast between Figure 4 and Figure 5 makes it clear that the number of HT tags available can also reduce concurrency and constrain performance. Figure 5 clearly indicates that even a relatively small cache can sustain most of the network performance, as long as there are not artificial constraints on concurrency.

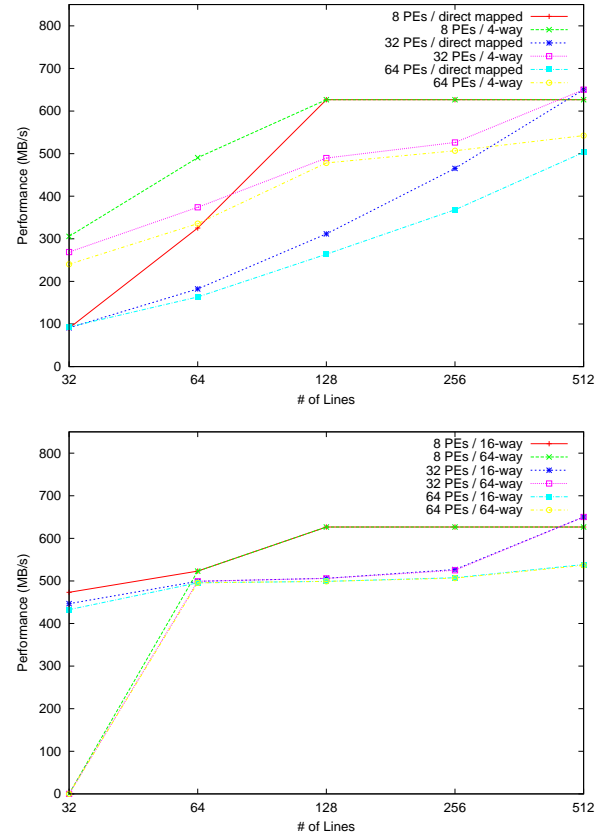
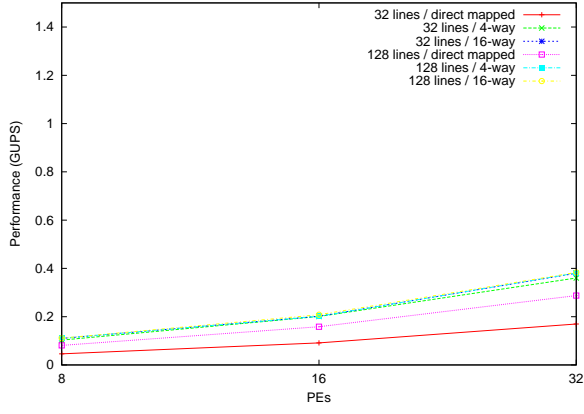


Fig. 5. The performance of an atomic unit using a cache with increased HT tags

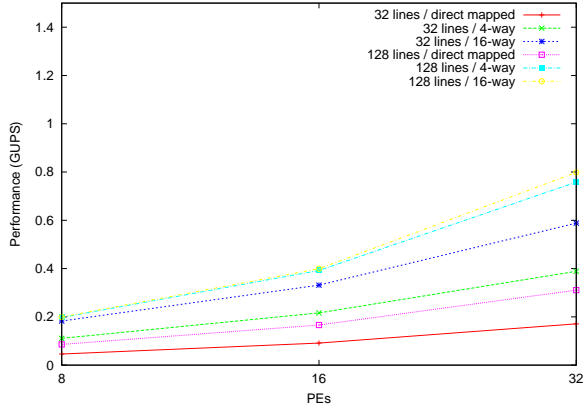
B. RandomAccess Performance

The RandomAccess benchmark has very different properties from the atomic benchmarks created for this study. Whereas the atomic benchmark mimics a small number of widely used variables, the RandomAccess benchmark is true to its name and accesses random locations on all of the nodes. Due to these properties, atomic updates from the RandomAccess benchmark never hit in the cache.

Figure 6 shows the performance of the RandomAccess benchmark when using (a) 32 and (b) 64 HT tags. In addition, the graphs show two sizes of caches (32 and 128 lines) as well as three associativities (direct mapped, 4-way, and 16-way). Since the RandomAccess benchmark cannot leverage cached data, the configurations explored were chosen for either their simplicity of implementation (32 lines, direct mapped) or their



(a)



(b)

Fig. 6. The performance of the RandomAccess benchmark using a cached atomic unit with (a) 32 HT tags and (b) 64 HT tags

success on the atomic benchmark in Section V-A. Unlike the benchmark used for Section V-A, the RandomAccess benchmark is sufficiently random that a 4-way cache is sufficient (4-way and 16-way configurations overlap). In addition, 128 lines is adequate to keep enough outstanding requests to HT. Once again, the need for 64 HT tags is evident, even though 32 tags is what is actually provided in an HT implementation.

Since the RandomAccess benchmark does not gain anything from the use of a cache, Figure 7 presents the performance for the RandomAccess benchmark when a cache is not present. In stark contrast to Figure 2, the RandomAccess benchmark suffers no performance penalty when a cache is not present.

Perhaps the most interesting, and most surprising, data is the comparison of the best simulated RandomAccess benchmark using atomics and our previous work simply using SHMEM[15]. Figure 8 clearly indicates that the SHMEM-based approach outperforms the atomic unit-based approach. This is because of bandwidth constraints imposed by the atomic-based approach, as we will see in Section VI. Notably, however, the RandomAccess benchmark is substantially different from the simple atomic benchmark, because all of the targets are *also* injecting requests and the traffic is very load balanced. Note, however, the advantage for the SHMEM

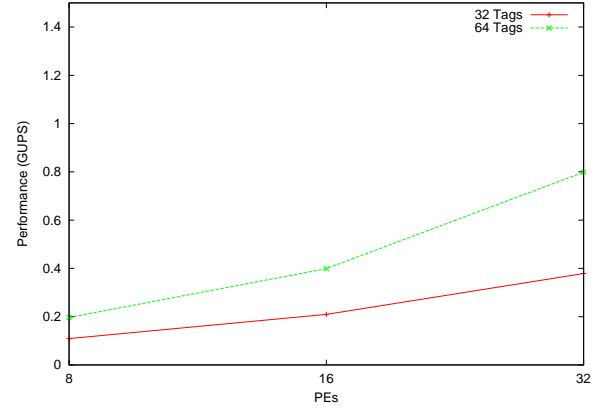


Fig. 7. The performance of the RandomAccess benchmark using an uncached atomic unit

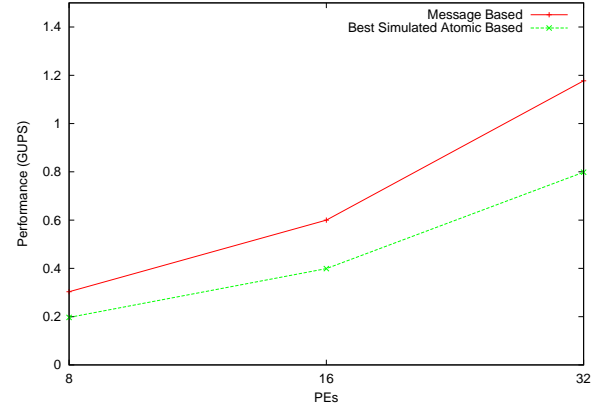


Fig. 8. Conventional algorithm compared to atomic unit based algorithm

approach in Figure 8 *only* occurs when the host memory interface has a sufficient performance advantage over the NIC interface to essentially overlap updates with network traffic arriving. It is also present *only* when the rate of small, SHMEM style messages is as high as the rate of atomic operations provided by the NIC.

VI. ANALYSIS

Several factors impact the RandomAccess performance results shown in Section V-B. While the cost of simulation prohibits the exploration of all conceivable factors that impact system performance, we can use analysis to explore the implications of some of the other system-level limitations. This section presents an analytical exploration of two major contributors to system performance.

A. Bandwidth-Based Comparison of NIC-Based RandomAccess to Baseline

As modeled, each command to the network interface requires 32 bytes of data to cross the HT interface. Each request (read or write) to the host requires 16 bytes, and each response from the host requires 12 bytes. Thus, the data that crosses the

interface between the NIC and the Host for each update when using the baseline (SHMEM message based) approach is:

$$NICtoHost = 16B(writeback) \quad (1)$$

$$HosttoNIC = 32B(command) \quad (2)$$

as the SHMEM-based approach is unidirectional from the NIC's perspective (i.e. commands are pushed from the host to the NIC and messages from the NIC to the host. In contrast, the NIC must read the data to update from the host, which yields:

$$NICtoHost = 8B(readrequest) + 16B(writeback) \quad (3)$$

$$HosttoNIC = 32B(command) + 12B(readresponse) \quad (4)$$

This bandwidth requirement then constrains the rate at which a node can generate updates to:

$$RandomAccessRate \leq \frac{BW_{HT}}{\max(NICtoHost, HosttoNIC)} \quad (5)$$

Considering that we simulated dual-processor nodes (two PEs per node), the network bandwidth limit is modeled in Figure 9. Given the evolution of modern processors to typically have more memory bandwidth than I/O bandwidth, this will often be the primary constraint (as it is for the system modeled), though it is always important to consider the overall system balance. The results in Figure 9 are *only* meaningful for a memory bandwidth to I/O interface bandwidth ratio that is sufficient and for a network interface where small SHMEM style messages are as fast (message rate) as the hardware atomic unit.

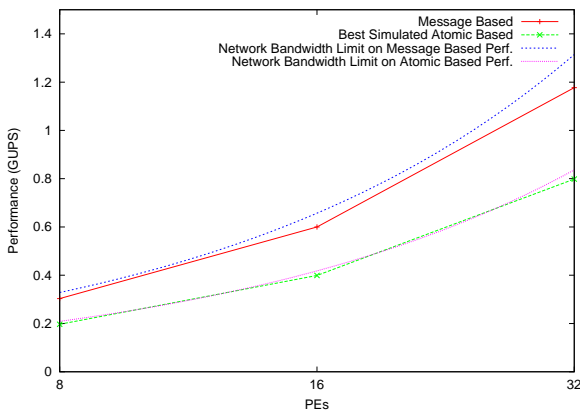


Fig. 9. Theoretical peaks for the RandomAccess benchmark

B. Outstanding Requests Required

As noted earlier, the high latency of the system modeled requires more outstanding requests than are typically used by

an HT node ID⁴. This is another substantial difference between the baseline and atomic-based approaches: the atomic-based approach performs reads *and* writes across the HT interface and therefore needs substantially more outstanding requests. The required number of requests (i.e. the HT “tags” referred to earlier) can be calculated as:

$$Requests = RandomAccessRate \times (RdLat + WrLat + MemLat) \quad (6)$$

Based on the size of requests and the data rate modeled, the *RandomAccessRate* for a single node is 52 million per second. For the latencies modeled (including a 70 ns memory latency), this requires the number of outstanding requests shown in Table III.

TABLE III
REQUESTS FOR RANDOMACCESS TRAFFIC BASED ON LATENCY

RdLat/WrLat	Request
500 ns / 250 ns	43
300 ns / 150 ns	27
100 ns / 50 ns	11

Note that the values in Table III are very much impacted by the traffic pattern. As seen earlier in Figure 2, a node that is only the target of atomic updates clearly needs more tags. This is because, for a node that is not generating traffic, the *HosttoNIC* traffic only consists of read responses (12 bytes). This means that the *AtomicAccessRate* in an equation analogous to Equation 6 would be over 95 million per second, and would actually be throttled by the network rate of 83 million per second. Thus, the requests needed are shown in Table IV. Comparing this theoretical calculation to the results in Figure 2, there is clearly a secondary limitation in the system performance as noted in Section V-A.

TABLE IV
REQUESTS AT THE TARGET FOR ATOMIC UNIT TRAFFIC BASED ON LATENCY

RdLat/WrLat	Request
500 ns / 250 ns	69
300 ns / 150 ns	44
100 ns / 50 ns	19

VII. CONCLUSIONS

The support for hardware atomic units on the network interface have the potential to bring interesting new capabilities to systems based on commodity processors. They have the potential to dramatically improve the performance of PGAS programming models, including languages such as UPC and Co-array Fortran, by providing the building blocks needed for a variety of synchronization operations. This paper has explored the hardware required to support two basic

⁴A single NIC can hypothetically have more than one node ID, but this does pose ordering challenges.

usage models: many nodes targeting shared synchronization variables and all nodes performing random updates across the machine — specifically, the HPCC RandomAccess (or GUPS) benchmark.

The results in this paper highlight four very important facts about the design of an atomic unit for the network interface. First, some level of caching is critical if reasonable performance is going to be achieved when targeting a single variable. Otherwise, there is not sufficient concurrency in the access stream. Second, when targeting an array of memory locations, the size and structure of the cache can be critical. A cache that is too small or has too little associativity can rapidly lead to substantially lower performance due to the conflicts in the cache that limit the achievable concurrency in the request stream to the host processor. Third, the access pattern of the RandomAccess benchmark *does not* require a cache, because it provides sufficient independent accesses to enable abundant concurrency over the HT interface; however, if a cache is provided, it too needs to have a reasonable size and associativity. Finally, in many scenarios, a hardware atomic unit on the NIC *will not* benefit the rate of system wide random accesses, because it requires more data to traverse the link to the host processor. While the use of an atomic unit in real workloads similar to the HPCC RandomAccess benchmark is likely to provide substantial benefits through things such as offloading work from the processor, such features are not captured in the benchmark and, therefore, are not apparent in this study.

VIII. FUTURE WORK

Atomic units on the network interface have been proposed for a variety of purposes. In addition to the types of operations discussed here, some in the community have requested floating-point operations on the network interface. In future work, we plan to study how floating-point atomic operations could be used for a variety of purposes, including accelerating collective operations.

REFERENCES

- [1] R. Alverson, “Red Storm,” in *Invited Talk, Hot Chips 15*, August 2003.
- [2] R. Brightwell, T. Hudson, K. T. Pedretti, and K. D. Underwood, “SeaStar interconnect: Balanced bandwidth for scalable performance,” *IEEE Micro*, vol. 26, no. 3, May/June 2006.
- [3] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg, “The Quadrics network: High-performance clustering technology,” *IEEE Micro*, vol. 22, no. 1, pp. 46–57, January/February 2002.
- [4] R. Brightwell, D. Doerfler, and K. D. Underwood, “A comparison of 4x InfiniBand and Quadrics Elan-4 technologies,” in *Proceedings of the 2004 International Conference on Cluster Computing*, September 2004.
- [5] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell, “A hardware acceleration unit for MPI queue processing,” in *19th International Parallel and Distributed Processing Symposium (IPDPS’05)*, April 2005.
- [6] K. D. Underwood, A. Rodrigues, and K. S. Hemmert, “Accelerating list management for MPI,” in *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, September 2005.
- [7] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, 1991.
- [8] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, “Introduction to UPC and language specification,” Tech. Rep. CCS-TR-99-157, May 1999.
- [9] R. W. Numrich and J. Reid, “Co-array Fortran for parallel programming,” *ACM SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, August 1998.
- [10] J. J. Dongarra, “The linpack benchmark: An explanation,” in *1st International Conference on Supercomputing*, June 1987, pp. 456–474.
- [11] “Top 500 web site,” November 2005. [Online]. Available: URL: <http://www.top500.org>
- [12] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, R. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi, “Introduction to the HPC challenge benchmark suite,” March 2005, <http://icl.cs.utk.edu/hpcc/pubs/index.html>.
- [13] S. Plimpton, R. Brightwell, C. Vaughan, K. Underwood, and M. Davis, “A simple synchronous distributed-memory algorithm for the HPCC RandomAccess benchmark,” in *2006 IEEE International Conference on Cluster Computing*, September 2006.
- [14] R. Garg and Y. Sabharwal, “Software routing and aggregation of messages to optimize the performance of the HPCC Randomaccess benchmark,” in *2006 ACM/IEEE International Conference for High-Performance Computing, Networking, Storage, and Analysis (SC’06)*, November 2006.
- [15] K. D. Underwood, M. J. Levenhagen, and R. Brightwell, “Evaluating NIC hardware requirements to achieve high message rate PGAS support on multi-core processors,” in *SC ’07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–10.
- [16] B. Nitzberg and V. Lo, “Distributed shared memory: A survey of issues and algorithms,” *Computer*, vol. 24, no. 8, pp. 52–60, 1991.
- [17] S. L. Scott, “Synchronization and communication in the T3E multiprocessor,” in *Seventh ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.
- [18] J. Beecroft, M. Homewood, and M. McLaren, “Meiko CS-2 interconnect Elan-Elite design,” *Parallel Computing*, vol. 20, no. 10-11, pp. 1627–1638, 1994.
- [19] Cray Research, Inc., *SHMEM Technical Note for C, SG-2516 2.3*, October 1994.
- [20] M. M. Michael and M. L. Scott, “Implementation of atomic primitives on distributed shared memory multiprocessors,” in *Proc. of the 1st IEEE Symp. on High-Performance Computer Architecture (HPCA-1)*, 1995, pp. 222–231.
- [21] Cray, Inc., “Cray X1E supercomputer,” <http://www.cray.com/products/systems/x1>.
- [22] D. Abts, A. Bataineh, S. Scott, G. Faanes, J. Schwarzmeier, E. Lundberg, T. Johnson, M. Bye, and G. Schwoerer, “The Cray Black Widow: A highly scalable vector multiprocessor,” in *Proceedings of the 2007 International Conference on High-Performance Computing, Networking, Storage and Analysis*, November 2007.
- [23] H. Hellwagner and A. Reinefeld, Eds., *SCI: Scalable Coherent Interface: Architecture and Software for High-Performance Compute Clusters*, ser. Lecture Notes in Computer Science. Springer, 1999, vol. 1734.
- [24] Infiniband Trade Association, <http://www.infinibandta.org>, 1999.
- [25] N. J. Boden, D. Cohen, R. E. F. A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, “Myrinet: A gigabit-per-second local area network,” *IEEE Micro*, vol. 15, no. 1, pp. 29–36, Feb. 1995.
- [26] D. Buntinas, D. K. Panda, and W. Gropp, “NIC-based atomic operations on Myrinet/GM,” in *First Workshop on Novel Uses of System Area Networks*, February 2002.
- [27] K. D. Underwood, M. Levenhagen, and A. Rodrigues, “Simulating Red Storm: Challenges and successes in building a system simulation,” in *21st International Parallel and Distributed Processing Symposium (IPDPS’07)*, March 2007.
- [28] D. Burger and T. Austin, *The SimpleScalar Tool Set, Version 2.0*, SimpleScalar LLC.